# Scalable, fault-tolerant matrix factorization

PINT 2018

Camille Coti

LIPN, CNRS UMR 7030, SPC, Université Paris 13

*May 2nd, 2018*

# Roadmap

# Scalability

**BIG** machines

- How do we program them?
- Need for **scalable** algorithms

Top 500, November 2017: 499 machines beyond $10^5$ cores

# Scalability

**BIG** machines

- How do we program them?
- Need for **scalable** algorithms

What is scalability?

- How does the algorithm evolve when we add cores
- Two dimensions
  - Operations
  - Communications

Top 500, November 2017: 499 machines beyond $10^5$ cores

# Scalability

**BIG** machines

- How do we program them?
- Need for **scalable** algorithms

What is scalability?

- How does the algorithm evolve when we add cores
- Two dimensions
  - Operations
  - Communications

Computations vs communications

- Nodes are **fast**
- Communication speed is limited by physical constraints

Top 500, November 2017: 499 machines beyond $10^5$ cores

Fetch one byte of data from DRAM

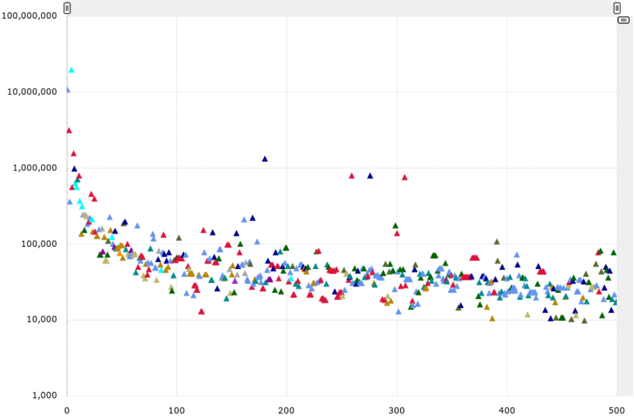
- 1988 : compute 6 flops
- 2004 : compute over 100 flops
- 2015 : compute 920 flops

Receive one byte of data from another processor

- 2015 : compute 4600 to $10^6$ flops

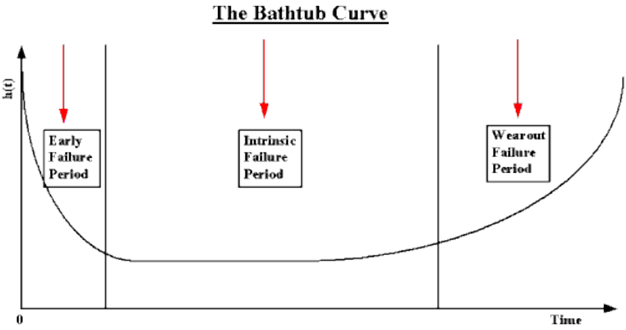
*source : Laura Grigori, SC15 keynote.*

# Top 500, November 2017



# Reliability of components

Life expectancy of an electronic component: the famous bathtub curve

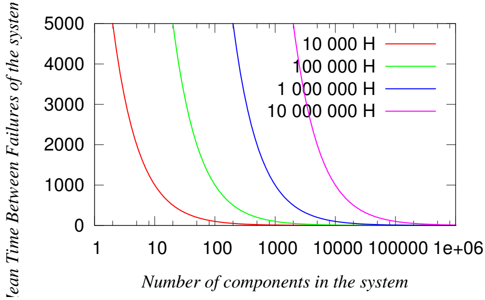


# Reliability of a distributed system

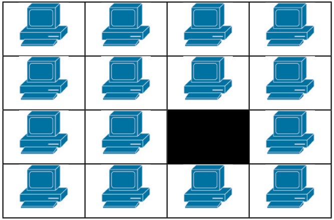
Mean Time Between Failures

$$MTBF_{total} = (\sum_{i=0}^{n-1} \frac{1}{MTBF_i})^{-1} \quad (1)$$

→ The more components a system is made of, the more likely it is to be hit by a failure.




At large scale, **failures cannot be dodged**



→ Must be tolerated.

In practice: Blue Waters, MTFB = 4,2H

# Requirements for algorithms

Therefore, algorithms must be:

- **Scalable**
  - Scale with the number of processes
- **Fault tolerant**
  - Able to survive beyond failures

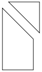
$\rightarrow$ Communication-avoiding algorithms

$\rightarrow$ User-Level Failure Mitigation for algorithm-based fault tolerance

# Los Tres Amigos

**QR factorization** : $A = QR$

- $R$ upper triangular
- $Q$ orthogonal



**LU decomposition** : $A = LU$

- $L$ lower triangular
- $U$ upper triangular

**Cholesky factorization** : $A = LL^T$

- $A$ symmetric, positive definite
- $L$ lower triangular

# QR factorization

Input matrix $A$

- Apply **unitary transformations** to the left: $R = U_\ell \dots U_1 A$
- Obtain the $Q$ using these transformations: $Q = U_1 \dots U_\ell$
- Unitary transformations: $U_i^H U_i = I$

Mostly used transformations:

- Givens rotation
- Householder reflections

Others: Cholesky-QR, Gram-Schmidt...

# QR factorization

Input matrix $A$

- Apply **unitary transformations** to the left: $R = U_\ell \dots U_1 A$
- Obtain the $Q$ using these transformations: $Q = U_1 \dots U_\ell$
- Unitary transformations: $U_i^H U_i = I$

Mostly used transformations:

- Givens rotation
- Householder reflections

Others: Cholesky-QR, Gram-Schmidt...

**Property**

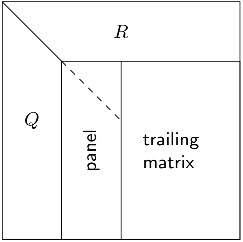
If $A = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix}$, $[R, Q] = \mathtt{qr}(A)$ and $[R, \hat{Q}] = \mathtt{qr}(\begin{pmatrix} R_1 \\ R_2 \end{pmatrix}) = \mathtt{qr}(\begin{pmatrix} \mathtt{qr}(A_1) \\ \mathtt{qr}(A_2) \end{pmatrix})$

**Tiled** factorization

- Perform operations on **blocks** rather than on scalars
- Can be Givens rotations performed on blocks
- Or Householder reflections spanning on less than the full column

# Communication-Avoiding QR

*Demmel, Hoemmen, Grigori, Langou 2007*



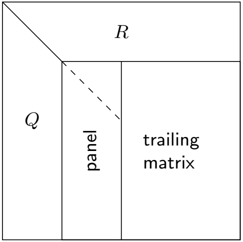

Works by **panels** :

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = Q_1 \begin{pmatrix} R_{11} & R_{12} \\ 0 & A^1_{22} \end{pmatrix}$$

Then, recursively, work on $A^1_{22}$...

# Communication-Avoiding QR

*Demmel, Hoemmen, Grigori, Langou 2007*




Works by **panels** :

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = Q_1 \begin{pmatrix} R_{11} & R_{12} \\ 0 & A^1_{22} \end{pmatrix}$$

Then, recursively, work on $A^1_{22}$...

## CAQR algorithm

1. Panel factorization:
   $\begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix} = Q_1 \begin{pmatrix} R_{11} \\ 0 \end{pmatrix}$
2. Compact representation:
   $Q_1 = I - Y_1 T_1 Y_1^T$
3. Update the trailing matrix:
   $(I - Y_1 T_1 Y_1^T) \begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix} = \begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix} - Y_1(T_1^T(Y_1^T \begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix})) = \begin{pmatrix} R_{12} \\ A^1_{22} \end{pmatrix}$
4. Continue recursively on the trailing matrix $A^1_{22}$

# Tall-and-Skinny QR

Panel factorization: cornerstone of the CAQR algorithm

$$\begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix} = Q_1 \begin{pmatrix} R_{11} \\ 0 \end{pmatrix}$$

The matrix $\begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix}$ is **tall and skinny** :

- number of lines $\gg$ number of columns

Specific algorithm to compute the QR factorization of a tall and skinny matrix: **TSQR**

# TSQR algorithm

Goal: compute the QR factorization of a matrix A:

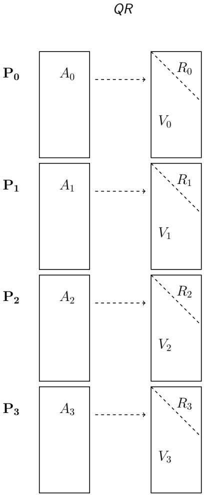
- $A = QR$
- $A$ is tall and skinny

To compute it in parallel on $P$ processes:

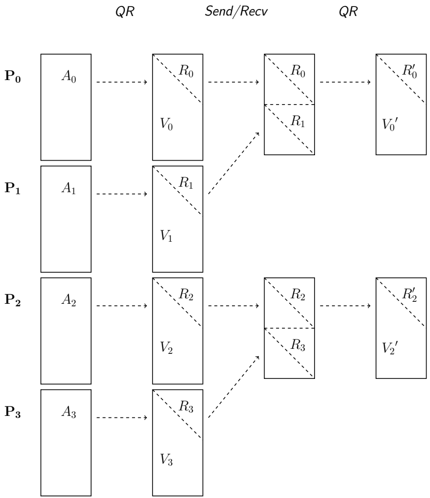
- $M$ = number of lines, $N$ = number of columns
- $M \geq NP$
  $\rightarrow$ at least square matrices on each process

$$\begin{pmatrix} A_1 \\ A_2 \\ A_3 \\ A_4 \end{pmatrix} = Q_1 \begin{pmatrix} R_1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

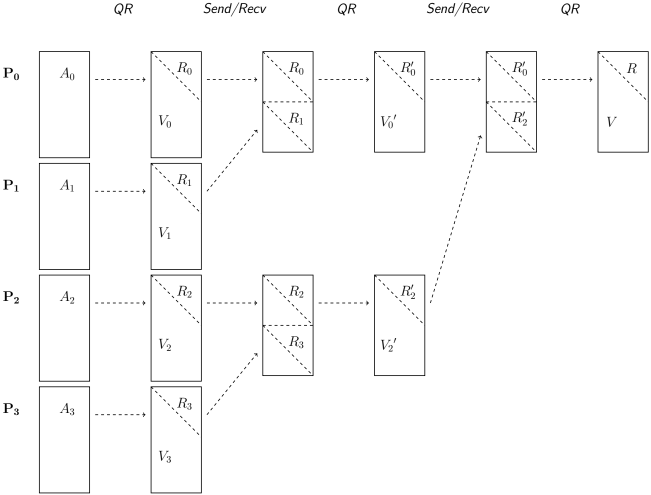
# TSQR


QR
$\mathbf{P_0}$
$A_0$
$R_0$
$V_0$
$\mathbf{P_1}$
$A_1$
$R_1$
$V_1$
$\mathbf{P_2}$
$A_2$
$R_2$
$V_2$
$\mathbf{P_3}$
$A_3$
$R_3$
$V_3$

# TSQR


*QR*
*Send/Recv*
*QR*
$\mathbf{P_0}$
$\mathbf{P_1}$
$\mathbf{P_2}$
$\mathbf{P_3}$
$A_0$
$A_1$
$A_2$
$A_3$
$R_0$
$V_0$
$R_1$
$V_1$
$R_2$
$V_2$
$R_3$
$V_3$
$R_0$
$R_1$
$R_2$
$R_3$
$R_0'$
$V_0{}'$
$R_2'$
$V_2{}'$

# TSQR


QR
Send/Recv
QR
Send/Recv
QR
$\mathbf{P_0}$
$\mathbf{P_1}$
$\mathbf{P_2}$
$\mathbf{P_3}$
$A_0$
$A_1$
$A_2$
$A_3$
$R_0$
$V_0$
$R_1$
$V_1$
$R_2$
$V_2$
$R_3$
$V_3$
$R_0$
$R_1$
$R_2$
$R_3$
$R'_0$
$V_0'$
$R'_2$
$V_2'$
$R'_0$
$R'_2$
$R$
$V$

# Update of the trailing matrix

Trailing matrix: denoted $C$, each $C_i$ being on process $i$.

$$C_i = \begin{pmatrix} C_i' \\ C_i'' \end{pmatrix} = \begin{pmatrix} C_i[: N-1] \\ C_i[N :] \end{pmatrix}$$

# Update of the trailing matrix

Trailing matrix: denoted $C$, each $C_i$ being on process $i$.

$$C_i = \begin{pmatrix} C_i' \\ C_i'' \end{pmatrix} = \begin{pmatrix} C_i[: N-1] \\ C_i[N :] \end{pmatrix}$$

Operation to perform:

$$\begin{pmatrix} R_0 & C_0' \\ R_1 & C_1' \end{pmatrix} = \begin{pmatrix} QR & C_0' \\ & C_0' \end{pmatrix} = Q \begin{pmatrix} R & \hat{C}_0' \\ & \hat{C}_1' \end{pmatrix}$$

# Update of the trailing matrix

Trailing matrix: denoted $C$, each $C_i$ being on process $i$.

$$C_i = \begin{pmatrix} C'_i \\ C''_i \end{pmatrix} = \begin{pmatrix} C_i[: N-1] \\ C_i[N :] \end{pmatrix}$$

Operation to perform:

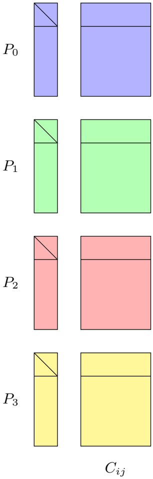
$$\begin{pmatrix} R_0 & C'_0 \\ R_1 & C'_1 \end{pmatrix} = \begin{pmatrix} QR & C'_0 \\ & C'_0 \end{pmatrix} = Q \begin{pmatrix} R & \hat{C}'_0 \\ & \hat{C}'_1 \end{pmatrix}$$

The compact representation becomes:

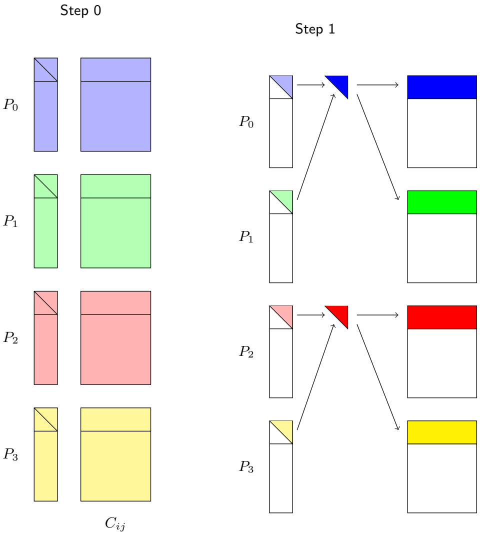
$$\begin{pmatrix} \hat{C}'_0 \\ \hat{C}_1 \end{pmatrix} = \left( I - \begin{pmatrix} I \\ Y_0 \end{pmatrix} T^T \begin{pmatrix} I \\ Y_1 \end{pmatrix}^T \right) \begin{pmatrix} C'_0 \\ C'_1 \end{pmatrix}$$

# Update of the trailing matrix: tree

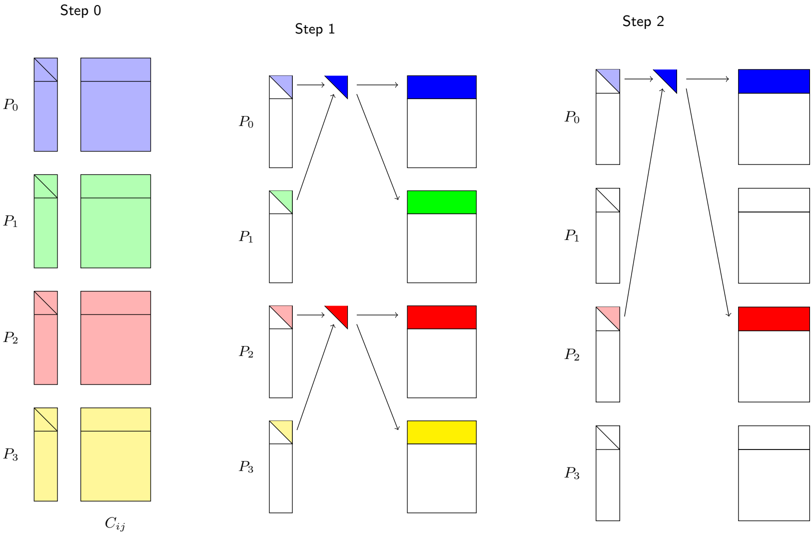
Step 0



# Update of the trailing matrix: tree



# Update of the trailing matrix: tree



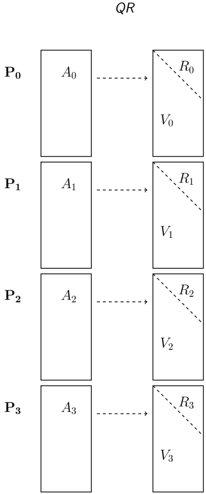
# Complexity

Matrix: $n \times n$ – process grid: $P = \sqrt{P} \times \sqrt{P}$ – block size: $b$

| | TSQR ($m \times n$, $P = P \times 1$) | CAQR | PDGEQRF |
|---|---|---|---|
| FLOPS | $\frac{4}{3}\frac{mn^2}{P} + \frac{2}{3}n^3 log(P)$ | $\frac{4}{3}\frac{n^3}{P} + \frac{4}{3}\frac{n^2 b}{\sqrt{P}} log(P)$ | $\frac{4}{3}\frac{n^3}{P}$ |
| Bandwidth | $\frac{3}{4}mn \cdot log(P)$ | $\frac{3}{4}\frac{n^2}{\sqrt{P}} log(P)$ | same |
| Latency | $log(P)$ | $\frac{5}{2}\frac{n}{b} log(P)$ | $\frac{3}{2}n log(P)$ |

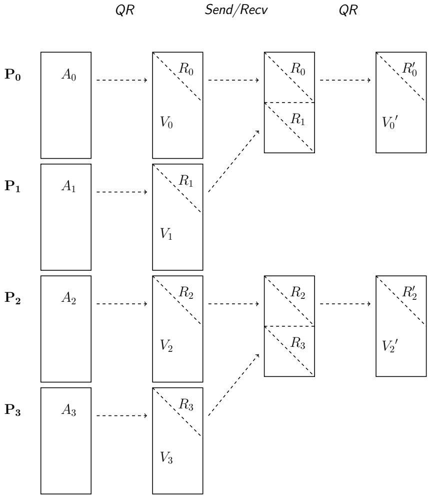
# Fault tolerant TSQR

Let's look at TSQR in details



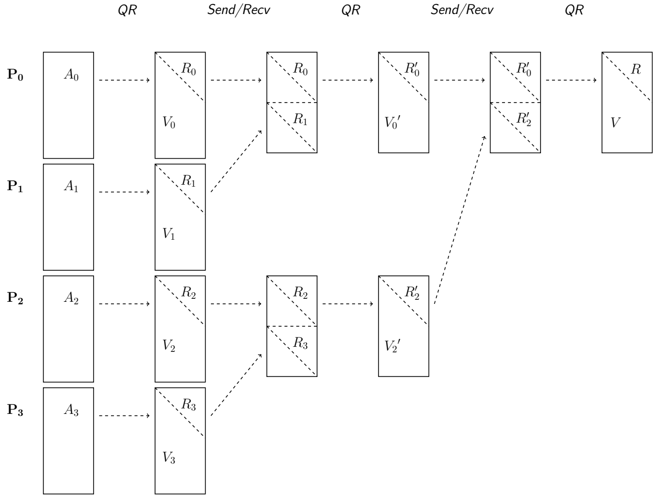
# Fault tolerant TSQR

Let's look at TSQR in details



# Fault tolerant TSQR

Let's look at TSQR in details



# Fault tolerant TSQR

Let's look at TSQR in details

- $P_0$ works beginning $\rightarrow$ end
- $P_2$ works during the first two steps, then stops
- $P_1$ and $P_3$ work during the first step, then stops

Let's put these lazy dudes to work!

# What do we expect from fault tolerance?

Have **one result** at the end

- No matter how many processes survive, one of them has the final answer
- Here: *Redundant TSQR*

# What do we expect from fault tolerance?

Have **one result** at the end

- No matter how many processes survive, one of them has the final answer
- Here: *Redundant TSQR*

Have the result **on a given process** at the end

- No matter how many processes survive, the one we want has the final answer
- Here: *Replace TSQR*

## What do we expect from fault tolerance?

Have **one result** at the end

- No matter how many processes survive, one of them has the final answer
- Here: *Redundant TSQR*

Have the result **on a given process** at the end

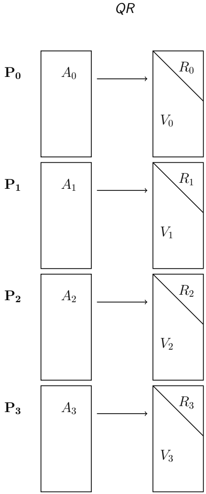
- No matter how many processes survive, the one we want has the final answer
- Here: *Replace TSQR*

Have the result on the expected process and **all the processes are alive**

- Finish with a system that looks as if nothing bad happened
- Here: *Self-Healing TSQR*

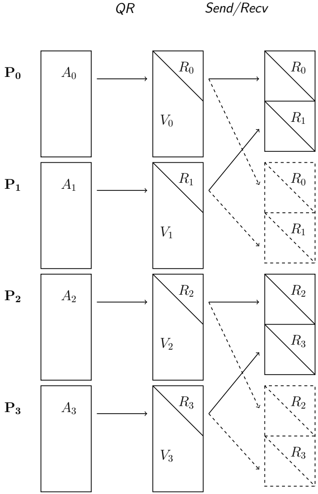
# Fault Tolerant TSQR: redundant TSQR 1/2

Introduce redundancy between processes: exchange between pairs.



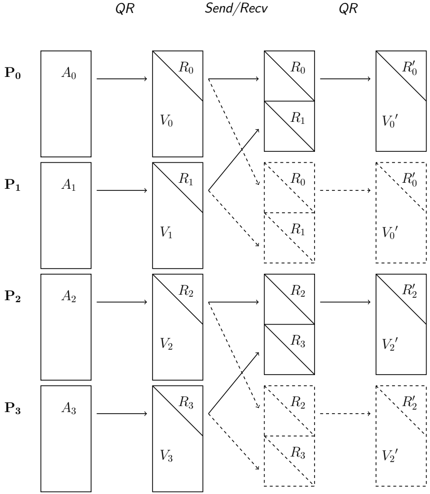
# Fault Tolerant TSQR: redundant TSQR 1/2

Introduce redundancy between processes: exchange between pairs.



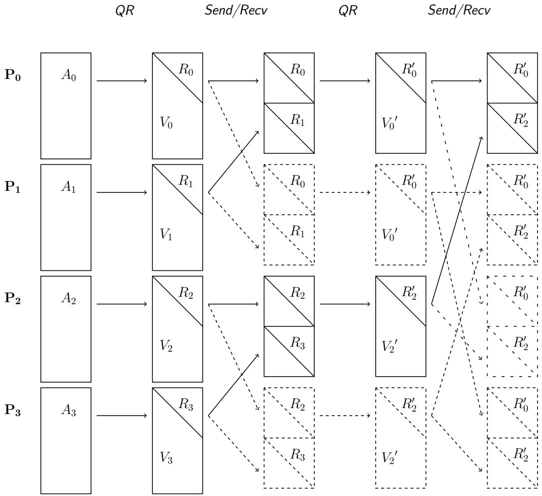
# Fault Tolerant TSQR: redundant TSQR 1/2

Introduce redundancy between processes: exchange between pairs.



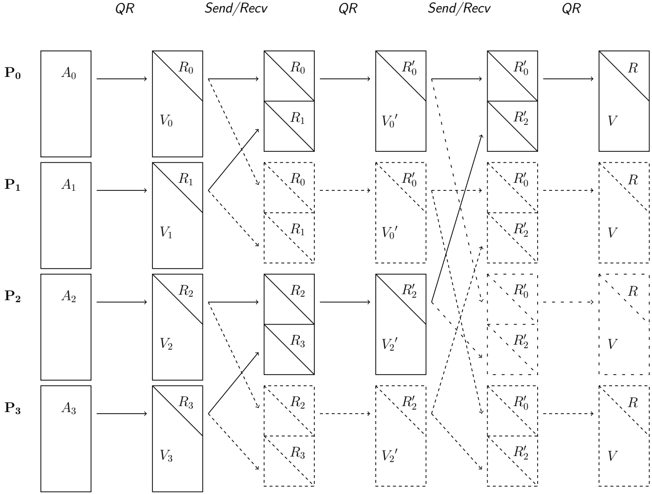
# Fault Tolerant TSQR: redundant TSQR 1/2

Introduce redundancy between processes: exchange between pairs.



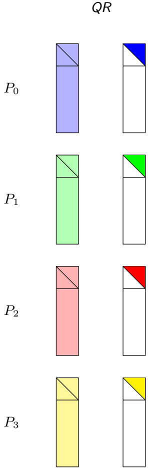
# Fault Tolerant TSQR: redundant TSQR 1/2

Introduce redundancy between processes: exchange between pairs.



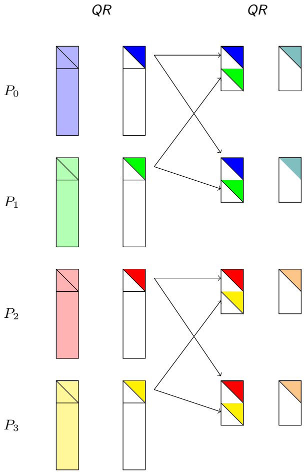
# Fault Tolerant TSQR: redundant TSQR 2/2

At each step, the redundancy rate is multiplied by 2.



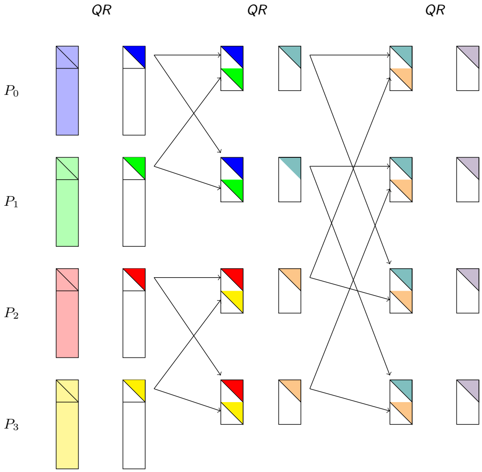
# Fault Tolerant TSQR: redundant TSQR 2/2

At each step, the redundancy rate is multiplied by 2.



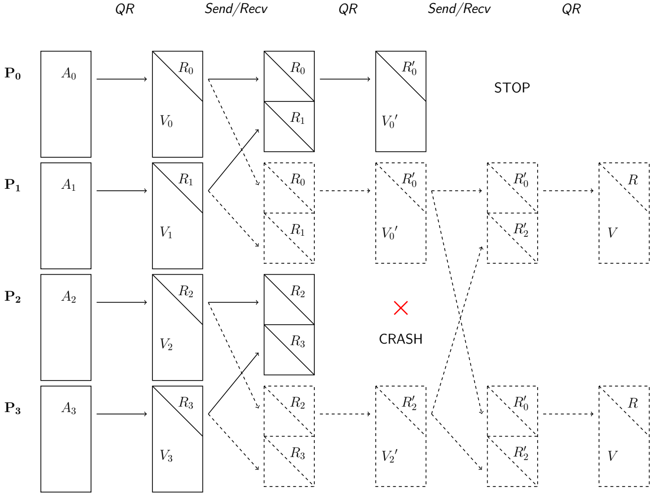
# Fault Tolerant TSQR: redundant TSQR 2/2

At each step, the redundancy rate is multiplied by 2.



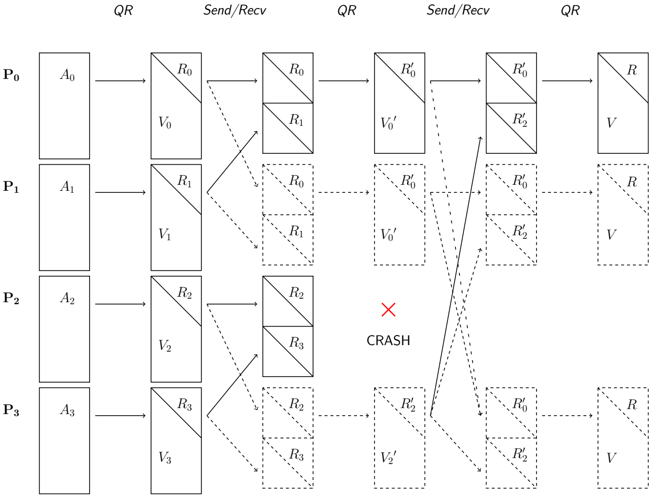
# Redundant TSQR: failure

If a process fails: the other ones can continue, except those who need to communicate with the failed process.



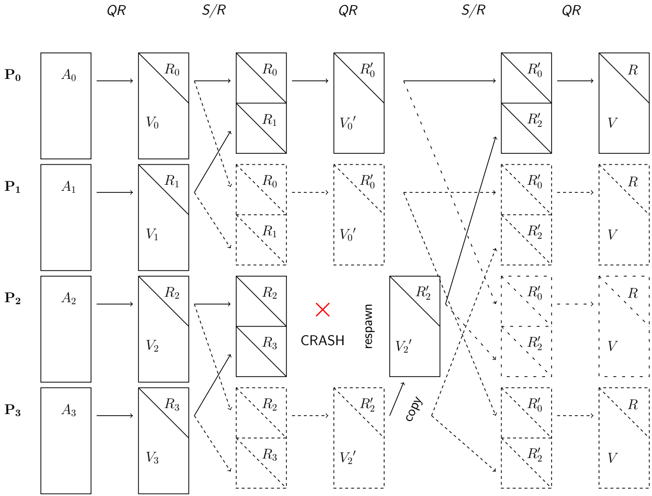
# Fault Tolerant TSQR: Replace TSQR

When a process fails, another one takes its place: $P_1$ acts as $P_2$.



# Fault Tolerant TSQR: Self-healing TSQR

Spawn a new process that recovers the data from a twin process

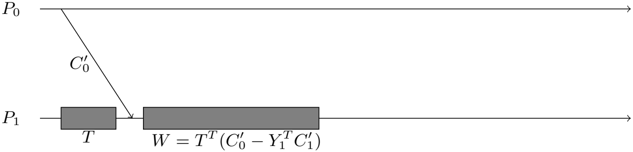


1. Introduction
   - Large-scale systems
   - Reliability
   - Los Tres Amigos
2. CAQR
   - QR factorization
   - Communication-avoiding QR
   - Complexity
3. FT-CAQR
   - FT-TSQR
   - FT-CAQR
4. Performance
5. Conclusion

# Update of the trailing matrix: algorithm

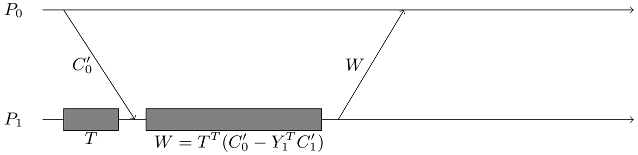

1. $P_0$ sends its $C'_0$ to $P_1$ while $P_1$ computes $T$

# Update of the trailing matrix: algorithm



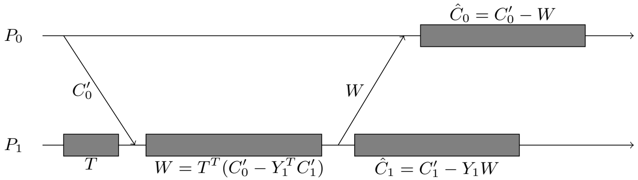

1. $P_0$ sends its $C_0'$ to $P_1$ while $P_1$ computes $T$
2. $P_1$ computes $W$

# Update of the trailing matrix: algorithm




1. $P_0$ sends its $C'_0$ to $P_1$ while $P_1$ computes $T$
2. $P_1$ computes $W$
3. $P_1$ sends $W$ to $P_0$

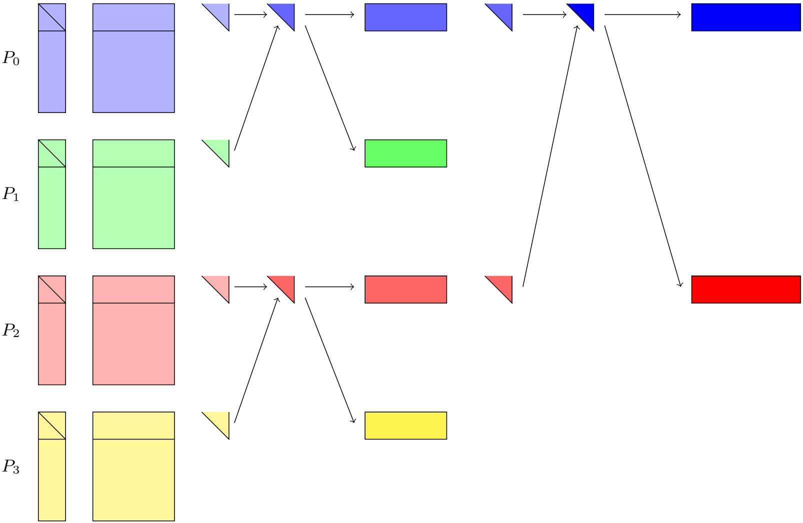
# Update of the trailing matrix: algorithm



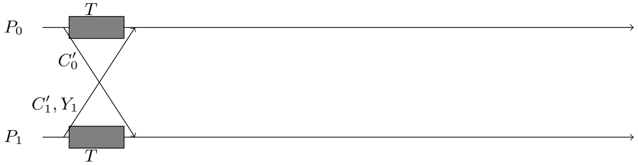

1. $P_0$ sends its $C'_0$ to $P_1$ while $P_1$ computes $T$
2. $P_1$ computes $W$
3. $P_1$ sends $W$ to $P_0$
4. $P_0$ computes $\hat{C}'_0$ and $P_1$ computes $\hat{C}'_1$

Continue... by pairs of processes.

# Update of the trailing matrix: tree

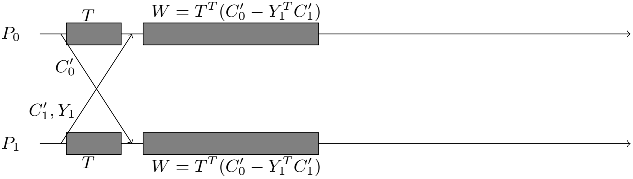


# Doing the pairwise computation on both processes



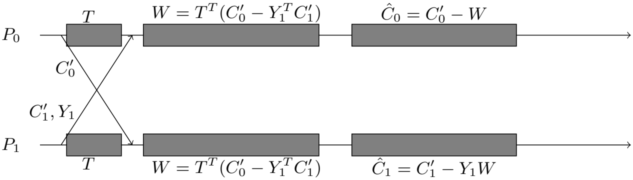

1. $P_0$ and $P_1$ exchange their $C'_i$, $P_1$ sends its $Y_1$

# Doing the pairwise computation on both processes




1. $P_0$ and $P_1$ exchange their $C_i'$, $P_1$ sends its $Y_1$
2. $P_0$ and $P_1$ both compute $W$

# Doing the pairwise computation on both processes



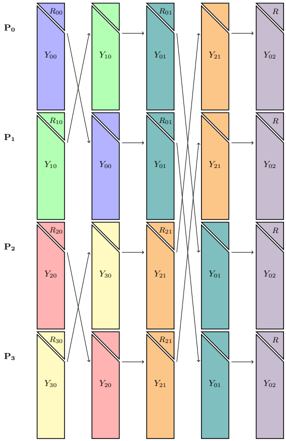

1. $P_0$ and $P_1$ exchange their $C'_i$, $P_1$ sends its $Y_1$
2. $P_0$ and $P_1$ both compute $W$
3. $P_0$ computes $\hat{C}'_0$ and $P_1$ computes $\hat{C}'_1$

Continue... by pairs of processes.

# Constructing the $Q$ matrix

The Q matrix is build using the Householder vector obtained from each local QR factorization

$\rightarrow$ Processes accumulate the HH vectors.



# Failure recovery

At the end of a given step, between $P_i$ and $P_j$:

- $P_j$ has $R_i$
  - if $P_i$ fails, $P_j$ can provide $R_i$ (already computed!)
- $P_i$ has $W$, $T$, $C'_i$, $C'_j$, and $\hat{C}'_i$;
  - if $P_j$ fails, $P_i$ can send sufficient data for any process that has $Y_j$ to recalculate $\hat{C}'_j = C'_j - Y_j W$
- $P_j$ has $W$, $T$, $C'_j$, $C'_i$ and $\hat{C}'_j$;
  - if $P_i$ fails, $P_j$ can recalculate $\hat{C}'_i = C'_i - W$
- $P_j$ has $Y_i$, $T_i$
  - if $P_i$ fails, $P_j$ can recalculate $Q_i$

# Performance evaluation

Performance evaluation: what do we measure?

- Overhead during fault-free execution
  - Very important!
  - Cost of the mechanisms put in place to make the FT possible
  - Here: additional communications
  - Same for the three algorithms

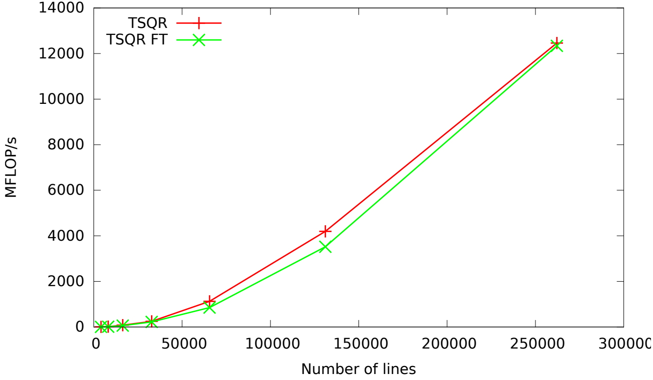
# Performance evaluation

Performance evaluation: what do we measure?

- Overhead during fault-free execution
  - Very important!
  - Cost of the mechanisms put in place to make the FT possible
  - Here: additional communications
  - Same for the three algorithms
- Recovery time
  - Depends on a lot of factors!
  - Failure detection (impossible with asynchronous communications)
  - Recovery made by the RTE (spawn and reconnect a new process)
  - Recovery protocol of the algorithm ← only interesting thing here, but hard to measure independently

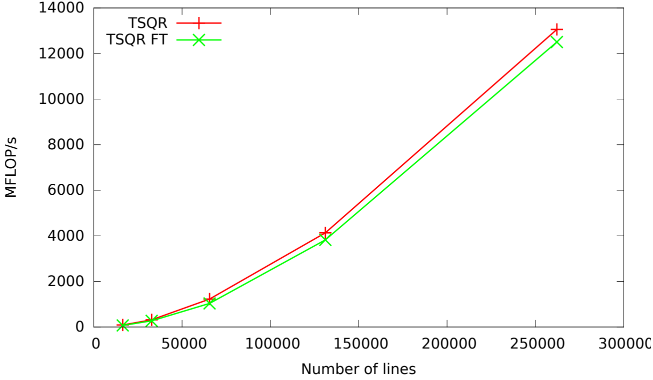
# Performance overhead on TSQR

64 processes, 64 columns ($P = 64$, $N = 64$)



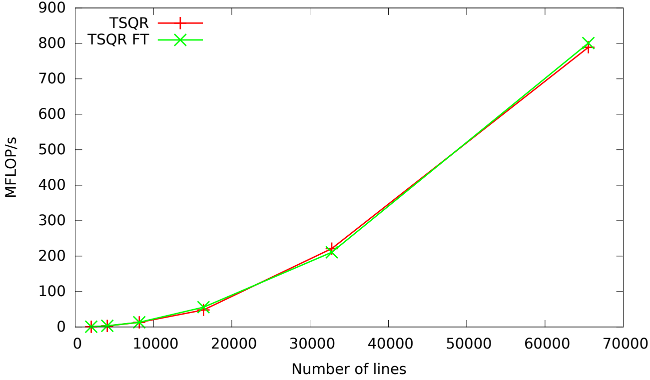
# Performance overhead on TSQR

256 processes, 64 columns ($P = 256,\ N = 64$)



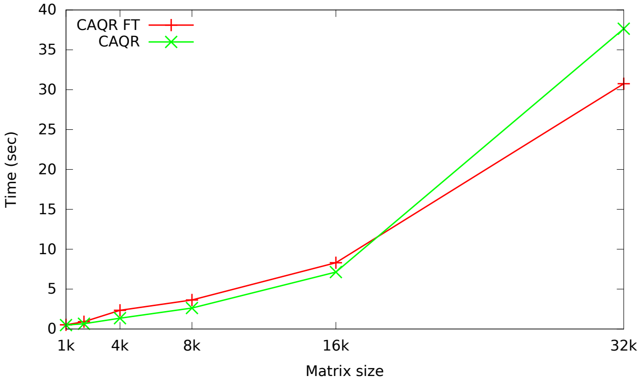
# Performance overhead on TSQR

16 processes, 128 columns ($P = 16$, $N = 128$)



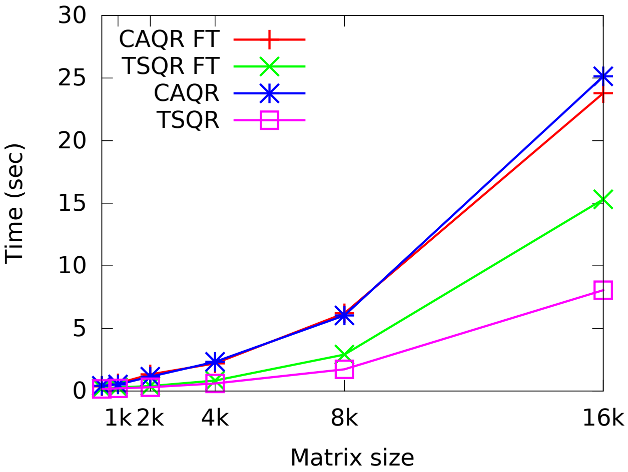
# Performance overhead on CAQR

256 processes, square matrices $(P = Q = 16,\ N = M)$



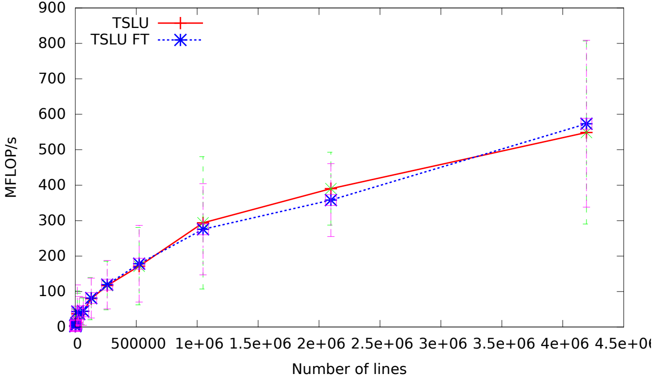
# Performance overhead on CAQR

64 processes, square matrices ($P = Q = 8$, $N = M$)



# Performance overhead on TSLU

256 processes, square matrices ($P = Q = 16$, $N = M$)



# Conclusion 1/2

Three protocols for fault-tolerant QR factorization of tall-and-skinny matrices

- Cornerstone for general QR factorization
- Three recovery algorithms, one for each semantics

Algorithm for FT update of the trailing matrix

- Fault-tolerant QR for general matrices

**Scalable FT protocol based on scalable algorithms**

Makes use of new features provided by the MPI-3 standard

- FT API now provided by MPI-3
- *User-Level Failure Mitigation*

Next step:

- Apply this to LU, Cholesky (the other *amigos*)
- Full performance analysis

## Conclusion 2/2

Algorithms based on **algebraic properties** of the computation

- Tiled algorithms
- Here: communications are minimized

Other operations

- Matrix-matrix multiplication: trivial, usual example
- Other ones?

Of **major importance** on current platforms

- Data locality: fit in cache (sequential, multicore, GPU...)
- Data locality: distributed memory
- Data locality: avoid data movements!!

**Fault tolerance**

- Intrinsic redundancy
- Algorithmic and algebraic properties
- Exploit data re-use

# References


- J. Demmel, L. Grigori, M. Hoemmen, & J. Langou: *Communication-avoiding parallel and sequential QR factorizations*, CoRR abs/0806.2159, 2008.
- J. Demmel, L. Grigori, M. Hoemmen & J. Langou: *Communication-optimal parallel and sequential QR and LU factorizations*, SIAM Journal on Scientific Computing 34 (1), 206-239, 2012.
- C. Coti: *Exploiting Redundant Computation in Communication-Avoiding Algorithms for Algorithm-Based Fault Tolerance*, IEEE HPSC 2016, New York, USA, April 2016.
- C. Coti: *Exploiting Redundant Computation in Communication-Avoiding Algorithms for Algorithm-Based Fault Tolerance*, CoRR abs/1511.00212, 2015.
- C. Coti: *Fault Tolerant QR Factorization for General Matrices*, CoRR abs/1604.02504, 2016.